A New Graph Data Structure

A Senior Honors Thesis

Presented in partial fulfillment of the requirements for graduation
with research distinction in Computer and Information Science in the undergraduate colleges of The Ohio State University

by

Brandon Sorg

The Ohio State University
June 2010

Project Advisor: Bruce Weide, Department of Computer Science and Engineering

The Ohio State University

Abstract

A New Graph Data Structure

by Brandon Sorg

Advisor Dr. Bruce Weide

A new data structure for graph representation has at least as good time efficiency as either of the two traditional data structures, the adjacency matrix and adjacency list, for any graph algorithm. For some algorithms, this new data structure actually produces a lower asymptotic bound than either of the other two. It is overall faster in practice as well and perhaps the best choice for general-purpose graph representation. A general-purpose sub-data structure is used to implement this new graph data structure and has applications of its own outside graph implementation.

Acknowledgments

I wish to thank my advisor, Dr. Bruce Weide, as well as all the members of the Europa Research Forum at The Ohio State University, who provided both the support and motivation to pursue this project and were the source of many of the ideas presented in this paper.

Table of Contents

Introduction

There are two standard data structures traditionally used to represent graphs: the adjacency matrix and the adjacency list. The decision as to which data structure to use depends on both the algorithm to be implemented and the properties of the graph on which the algorithm is run. There is thus an issue when implementing graph algorithms if the overall structure of the graph is not known a priori or if a certain problem requires the use of multiple algorithms, some of which may be better suited for one of the two traditional data structures. In this paper we shall introduce a new data structure that, asymptotically speaking, has at least as good time efficiency as either of these two traditional data structures for any graph algorithm. For some algorithms, this new data structure actually produces a lower asymptotic bound than either of the other two.

We will then compare the actual run-times of algorithms implemented using each of the various data structures and detail the advantages of each data structure. Since the data structure presented in this paper is asymptotically at least as good as the more traditional ones, it serves as a good choice to represent an abstract graph component. We will introduce a list of basic operations one would expect from a graph component, discuss its efficacy in implementing general graph algorithms, and compare the time efficiencies of the component operations when being represented by the different data structures.

While graph algorithms and data structures are the main focus of this paper, this new graph data structure can itself be viewed as simply an application of a more basic data structure. We will discuss the properties and implementation of this data structure as well as its own applications beyond graph algorithms.

Section I
The Partitionable Set

1.1: An Introduction to the Partitionable Set

Throughout Section 1, we will describe several variations on the same theme, which we here call the *partitionable set.* We will begin in Section 1.2 with the most fundamental of the variations and continue by augmenting the data structure with more functionality.

In general, the partitionable set is meant to be used in problems with the following properties:

1) We are working with a finite and known set of elements D
2) D is partitioned into disjoint subsets S_1, S_2, …, S_m

We also have the desire to perform the following operations as efficiently as possible:

1) Find an arbitrary element of some S_i
2) For any $x \in D$, find the integer i such that $x \in S_i$
3) Move any $x \in D$ from one subset S_i to another subset S_j

For example, we may want to keep track of a committee whose members are in at most one subcommittee. In this situation there may be movement of committee members from one subcommittee to another, and the designation of which members are in which subcommittee can

in no way be known in advance. We may want to be able to locate an arbitrary member of some given subcommittee, and we may also need to be able to determine which subcommittee a given member is in.

1.2: The Partitionable Set with Only One Interesting Subset

We first consider the case where our domain D is partitioned into only two subsets, S and S^c. For our purposes at the moment, we only really care about elements in S. In other words, we don't need to be able to quickly locate an element in S^c, but we still need to be able to locate an element in S and to determine whether a given x is in S or S^c. Also, for simplicity, our domain will be the integers $\{1, 2, \ldots, n\}$. We thus want the operations listed in Table 1.

Operation	Description
`Set_Domain(n)`	Sets domain D to $\{1, 2, \ldots, n\}$ with S empty
`Add_To_Subset(x)`	Adds an element $x \in S^c$ to S
`Remove_From_Subset(x)`	Removes an element $x \in S$ from S
`Find_Any_In_Subset(x)`	Produces a value for x such that $x \in S$
`Is_In_Subset(x)`	Returns a Boolean value whether $x \in S$
`Size_Of_Subset()`	Returns the integer value $\|S\|$

Table 1: The operations for the partitionable set with one interesting subset

Now, we want to minimize the run-time of each of these operations, and in fact we will describe a data structure that performs all the operations listed in Table 1 in constant time. We start with the general appearance of this data structure.

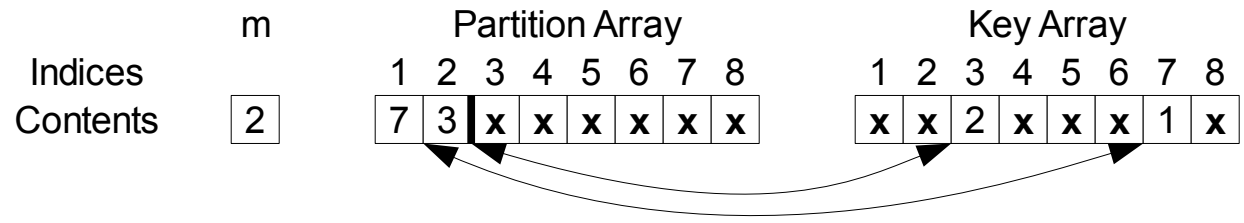

Figure 1: The partitionable set of 8 elements, with elements 3 and 7 in the subset S

The partitionable set, as shown in Figure 1, is made up of two arrays, *Partition* and *Key*. All the elements of *S* appear in the partition array on the left side of the partition line, as denoted by the integer *m*. Furthermore, for every index $i \leq m$ where $Partition[i] = j$, we also have that $Key[j] = i$. All other entries in the two arrays are meaningless and may take any arbitrary value, marked as **x** in Figure 1.

Now we shall show that the six operations in Table 1 can be performed in constant time.

`Size_Of_Subset()`

Naturally, the location of the partition line, denoted by *m*, is also the number of elements in *S*, so finding the size of the subset only requires returning that value.

`Find_Any_In_Subset(x)`

We can find an arbitrary element of S by looking at $Partition[m]$.

`Is_In_Subset(x)`

To test whether $x \in S^c$ or $x \in S$, we look first at whether $1 \leq Key[x] \leq m$. If we find that $Key[x]$ is in the valid range, we check if $x = Partition[Key[x]]$ and return true if and only if both criteria hold true. To prove this, suppose $x \in S$. Then $Key[x]$ must be the index of some entry in the partition array to the left of the partition line, so $1 \leq Key[x] \leq m$. By design, $Partition[Key[x]] = x$.

Now suppose $x \in S^c$. This means the contents in the key array at index x are meaningless and could take on any arbitrary value. If $Key[x] < 1$ or $Key[x] > m$, we are done and can return false. Otherwise, $Key[x]$ corresponds to an index in the partition array containing an element $y \in S$. Thus, $Partition[Key[x]] = y \neq x$ and we return false.

`Set_Domain(n)`

Immediately upon initializing the domain of the partitionable set to be of size n, S will be empty. The partition and key arrays will thus contain only meaningless entries. Since we do not care what the contents of the arrays are, the only thing thing that needs to be initialized is m, and we can create uninitialized arrays of arbitrary length in constant time. It should be noted that in practice we may observe that creating larger arrays, even uninitialized ones, will sometimes

require more time. This is due to the physical constraints of the computer and its ability to allocate enough available memory. We will for the rest of this paper ignore this difference and assume that the needed memory is readily available.

Unlike the four previous operations discussed, the two remaining operations in Table 1, adding and removing an element from *S*, change the contents of the data structure. For these two we need to show not only that they take place in constant time but also that they do not destroy the properties of the structure that make the previous operations possible to do in constant time. For these, we provide both the pseudocode and an explanation.

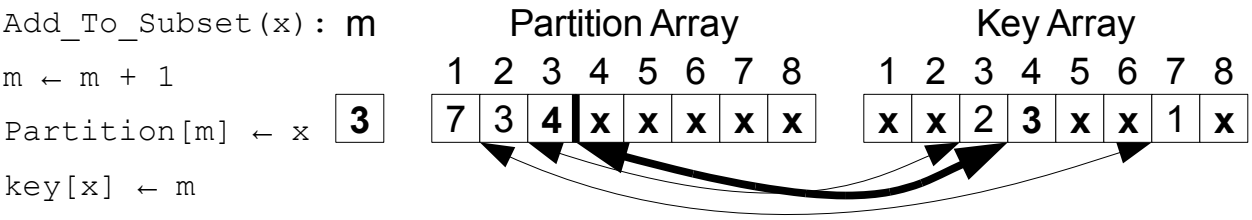

Figure 2: Figure 1 after adding element 4 to S

To add an element, we slide the partition line to the right one spot by incrementing *m*, and then we fill in the once unused space that used to be *Partition*[*m*+1] and *Key*[*x*]. Since $x \in S^c$ at the beginning of the operation, the contents at *Key*[*x*] are meaningless. Similarly, the contents of *Partition*[*m*+1] are in the beginning meaningless, so we can pair these two entries up in the same way that the other entries in the arrays are paired, and since the previous pairs are unaffected, the necessary structure is maintained.

```
Remove_From_Subset(x):
swap(key[x], key[partition[m]])
swap(partition[key[x]], partition[m])
m ← m − 1
```

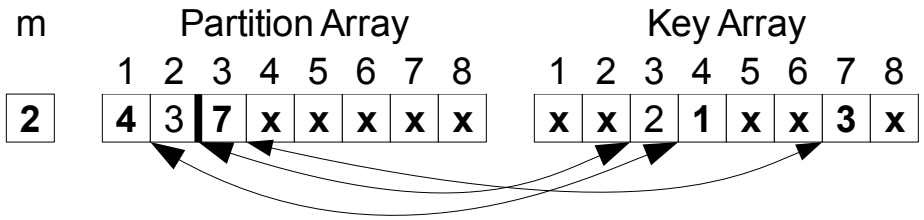

Figure 3: Figure 2 after removing element 7 from S

The swap functions serve to move the contents of the partition array corresponding to element x up to the partition line while maintaining the property that the partition array and the key array are related as before. Now that the element we wish to remove is next to the partition line, we can slide the partition line to the left. From now on we can ignore the contents of what is now *Partition*[*m*+1] and *Key*[*Partition*[*m*+1]], just as we would normally do. In Figure 3, for example, the 7 located at index 3 of *Partition* and the 3 located at index 7 of *Key* can now be viewed as *x*. We do not need to do anything explicitly to ignore these values, but rather from now on we can treat them as being arbitrary.

We have hence described in full the data structure that performs in constant time all the operations listed in Table 1.

1.3: The Partitionable Set with Many Interesting Subsets

We now consider the case where we have several different disjoint subsets S_1, S_2, …, S_k in our domain D. However, as in Section 1.2, we are still not necessarily concerned with all of D. In other words, $S_1 \bigcup S_2 \bigcup \ldots \bigcup S_k$ will in general not equal D. In fact, upon initialization, $S_1 \bigcup S_2 \bigcup \ldots \bigcup S_k$ will be empty. We now want a data structure that handles the operations listed in Table 2.

Operation	Description
`Set_Domain(n,k)`	Sets domain D to $\{1, 2, \ldots, n\}$ with $(S_1 \bigcup S_2 \bigcup \ldots \bigcup S_k)$ empty
`Add_To_Subset(x,i)`	Adds an element $x \in (S_1 \bigcup S_2 \bigcup \ldots \bigcup S_k)^c$ to S_i
`Remove_From_Subset(x,i)`	Removes an element $x \in S_i$ from S_i
`Find_Any_In_Subset(x,i)`	Produces a value for x such that $x \in S_i$
`Change_Subset(x,i,j)`	Moves an element $x \in S_i$ to S_j
`Is_In_Subset(x,i,j)`	Returns a Boolean value whether $x \in S_i \bigcup S_{i+1} \bigcup \ldots \bigcup S_j$
`Size_Of_Subsets(i,j)`	Returns the integer value $\lvert S_i \bigcup S_{i+1} \bigcup \ldots \bigcup S_j \rvert$

Table 2: The operations for partitionable set with k interesting subsets

When extending the functionality of the partitionable set in Section 1.2 to be able to handle the operations in Table 2, we can keep the same same relation between the arrays *Partition* and *Key* as before. We only need to examine the changes in *Partition* and *m*. Figure 4 shows an example of how these two may appear.

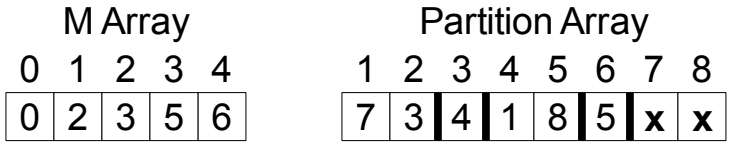

Figure 4: M and Partition of a partitionable set with 4 subsets

Instead of a single integer *m*, we now have an array *M* of *k* integers. The i^{th} integer in M denotes the location of the i^{th} partition line, which separates the S_i and the S_{i+1} portions of Partition. `Find_Any_In_Subset(x,i)` will now require looking at *Partition*[*M*[*i*]]. The operation `Size_Of_Subsets(i,j)` can then be performed in constant time by taking *M*[*j*] – *M*[*i*–1], and `Is_In_Subset(x,i,j)` can be performed by checking if *Partition*[*x*] falls in the interval [*M*[*i*–1] + 1, *M*[*j*]] and then checking the corresponding entry in *Key*. `Set_Domain(n,k)` is the same as before, but now we have to initialize the *M* to contain all 0's, taking *O*(*k*) time.

The remaining operations in Table 2 pose a bit of a problem. Each subset appears in *Partition* as a contiguous sequence of integers. Inserting into a specified subset requires there to be an empty space available next to this sequence, which may not exist, as in subsets 1 through 3 in Figure 4. Removing an element creates an undesirable hole in the array. Fortunately, we do not have to shift the entire array over by one in order to create or fill a space. Figure 5 shows the steps taken in *Partition* to move element 3 in Figure 4 from subset 1 to subset 4. We swap the specified

element to the end of the section of the array denoting the current subset it is in (adjusting *Key* accordingly) and then slide the partition line over by one so that the specified element is now in the next subset over. We repeat until the desired subset is reached. Essentially, we call a version of `Remove_From_Subset(x)` from Section 1.2 for each subset standing between the origin and the goal. Thus, a call to `Change_Subset(x,i,j)` will take $O(|i-j|)$ time. The operations `Remove_From_Subset(x,i)` and `Add_To_Subset(x,i)` will behave similarly and take $O(k-i)$ time.

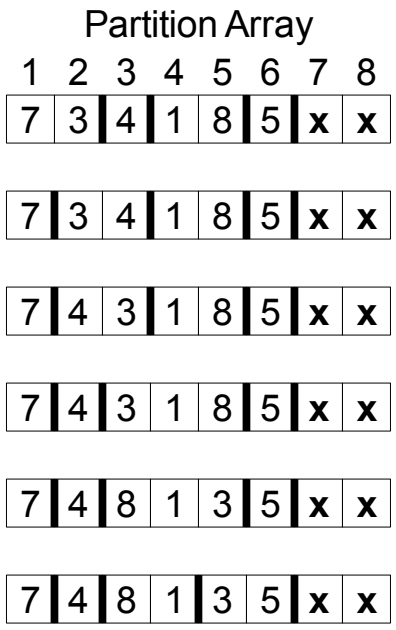

Figure 5: Changing 3 from S1 to S4

In going from one subset to multiple subsets, we have in a sense lost the ability to run all the operations we want in constant time. However, for some purposes we may have k as a small, fixed constant, in which case all the above operations do indeed operate in constant time with respect to n. There may be problems where we would not want to view the value of k as fixed, and we could even extend this data structure to accommodate changing the value of k at any time by adding or removing subsets, but with the asymptotic run-times given, this is perhaps not the best use of the partionable set. Thus, both practically and theoretically, it is perhaps best to view k as fixed.

We may desire more operations than the ones listed in Table 2. For example, we may desire an operation that, given $x \in D$, returns the value i such that $x \in D$. For this operation, we can look at $Key[x]$ and locate the i such that $M[i-1] < Key[x] \leq M[i]$. Since M is ordered, we can do this by interval halving, taking $O(\log(k))$ time. Similarly, we could also do interval halving with the `Is_In_Subset(x,i,j)` operation until we get to i.

We may also want to be able to take unions between subsets. In general, we cannot do so easily. However, if we want to take the union of S_i and the S_{i+1}, we can do so in constant time by setting the partition line in *Partition* between the two sets equal to one of the partition lines on either side of the two. We now view one of S_i or S_{i+1} as being the union of the two, while the other is now empty. This may be useful if we know a priori the order in which we will be taking unions.

Finally, if want to list off all the elements of a specific subset, an iterator may be very useful. To iterate through all of S_i, we create an integer p so that initially $p = M[i-1]$. To return the next element in S_i, we increment p and then return $Partition[p]$. We repeat in order to list all the elements of S_i until $p = M[i]$.

Section 1.4: The Partitionable Set, All of Whose Subsets Are Interesting

While the previous two models for partitionable set essentially do partition the entire domain D, one of the sets in the partition lacks the same functionality as the others. As described in Section

1.3, the partitionable set contains k interesting subsets $S_1, S_2, \ldots, S_k$, with $(S_1 \bigcup S_2 \bigcup \ldots \bigcup S_k)^c$ not necessarily empty, but while $(S_1 \bigcup S_2 \bigcup \ldots \bigcup S_k)^c$ is a subset in the partition, we cannot use `Find_Any_In_Subset(x,i)` on it. We now want to add this functionality. Hence, we want to have the operations listed in Table 3.

Operation	Description
`Set_Domain(n,k)`	Sets domain D to $\{1, 2, \ldots, n\}$ with $S_0 = D$ and k – 1 empty subsets S_1 through S_{k-1}
`Find_Any_In_Subset(x,i)`	Produces a value for x such that $x \in S_i$
`Change_Subset(x,j,i)`	Moves an element $x \in S_j$ to S_i
`Is_In_Subset(x,i,j)`	Returns a Boolean value whether $x \in S_i \bigcup S_{i+1} \bigcup \ldots \bigcup S_j$
`Size_Of_Subsets(i,j)`	Returns the integer value $\lvert S_i \bigcup S_{i+1} \bigcup \ldots \bigcup S_j \rvert$

Table 3: The operations for partitionable set with every set interesting

We could easily implement the operations in Table 3 using the data structure described in Section 1.3 with k interesting elements, but at first glance this means the `Set_Domain(n,k)` operation would require adding all n elements in D to S_0, taking $O(n)$ time. However, we do not really need to explicitly place all n elements into S_0 at initialization. We only need to make sure the other four operations function the way they should.

To get all the operations in Table 3 to run in constant time, we implement our new set of operations with a partitionable set from Section 1.3 using k interesting subsets. In other words,

we want to implement our partitionable set on domain D with subsets $S_0, S_1, \ldots, S_{k-1}$ with operations from Table 3, and to do so we will use the variant of partitionable set on D with subsets $S'_1, S'_2, \ldots, S'_k$ that has operations from Table 2. We view S_1 through S_{k-1} the same way we would under the partitionable set model from Section 1.3, so $S_i = S'_i$ for all $i \in \{1, 2, \ldots, k-1\}$. Operations concerning only subsets S_1 through S_{k-1} should therefore be implemented using the corresponding operation on the subsets S'_1 through S'_{k-1}. Let $T = (S'_1, S'_2, \ldots, S'_k)^c$. S_0 is then the disjoint union $S'_k \bigcup T$. Specifically, S'_k is the set of initialized elements of S_0 and T is the set of uninitialized elements of S_0. We can thus run `Set_Domain(n,k)` in $O(k)$ time by initializing the partitionable set on the k interesting subsets S'_1 through S'_k and then adding element 1 to S'_k in order to ensure S'_k is nonempty. Since we can check in constant time if an element is in T by checking if it is in S'_1 through S'_k, the implementation of `Is_In_Subset(x,i,j)` follows accordingly. Similarly, the implementation for `Size_Of_Subsets(i,j)` is also straightforward.

To implement `Change_Subset(x,i,j)` for $j = 0$, we only need to change x to be in S'_k. For $i = 0$, we check first if $x \in S'_k$ or $x \in T$. Then we perform the corresponding operation from Table 2 to change x to be in S_j. `Find_Any_In_Subset(x,i)` will call the corresponding find operation on S'_k if S'_k is nonempty. However, the operation is not so easy if S'_k is empty. We thus want to make sure S'_k is always nonempty.

To do so, we make a small addition to `Change_Subset(x,i,j)`. Every time we call `Change_Subset(x,0,j)`, if $x \in S'_k$, then if $x+1 \in T$, we add $x+1$ to S'_k. This means that, if we examine *Key*, contiguous blocks with indices i through j such that $j+1 \in S_0$ and for each $p \in \{i, i+1, \ldots, j\}$ $p \notin S_0$, then we have $j+1 \in S'_k$.

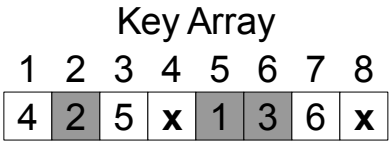

Figure 6: Key Array for partitionable Set of 8 elements. Indices $\notin S_0$ are gray

Figure 6 shows an example of *Key*, where index 1 and every index that comes after a contiguous block of elements not in S_0 is itself in S'_k. Thus, if S'_k is empty, the set of such contiguous blocks must include the entire domain $\{1, 2, \ldots, n\}$, but that would mean S_0 is empty and `Change_Subset(x,0,j)` should not have been called. If we want an iterator to be able to list all the elements of S_0, we would similarly have to change the iteration process to add $x+1 \in T$ to S'_k if we find element x. Since contiguous blocks in Key corresponding to elements in S_0 must begin with an element in S'_k, this iteration process will indeed list all the elements of S_0.

Section 1.5: More on Partitionable Set

In solving practical problems using the partitionable set data structure, we probably want to keep track of more information than just which elements are in which subsets. For example, if we are working with a set of n elements, we might want to have an array *Contents* of length n that stores whatever extra information about each element is needed for the problem on which we are working. There does not need to be anything special about this array, but the partitionable set

allows us to work with this array just as we normally would but without having to initialize the elements of the array. We can check to see if an element i is in one of the initialized subsets in the partitionable set before trying to access *Contents*[i]. Using this uninitialized array does not provide a benefit to the overall running time of the algorithm if we end up using all of the array so that all the elements get initialized at some point anyway. However, if we expect that we will only use $o(n)$ of the array, then we will see gains in time efficiency in exchange for extra memory costs.

We may also want to have first-in, first-out behavior when adding and removing elements from one of the interesting subsets (a behavior that the partitionable set does not have by itself). To do this, we could make a linked list for each of the interesting subsets of the partitionable set. The elements in the linked list will be precisely the elements of the corresponding subset, so that with the combination of partitionable set and linked list, we can keep a first-in, first-out structure but also have the ability to determine of membership of specified elements in specified linked lists in constant time. We could also keep an array of pointers, so that at *Contents*[x] we find the location where x is in the linked list. Hence, we have a first-in, first-out structure with the added ability to find elements in the middle of the list in constant time. This again comes at the cost of more memory. If there are n possibilities of elements to be in the list, then we would need $O(n)$ memory, but the actual number of elements in the linked list itself could be much smaller.

One might wonder by now where specifically this absence of initialization of the arrays can be advantageous. In general, the method of using the partitionable set as described above could be useful whenever we have a bound on the number of possible elements we can have in some data structure, but we expect a much smaller number to actually be present or interesting. For a simple example, consider matrix multiplication. If we are multiplying with a matrix with many zeroes, we would probably want to be able to skip over the zero elements, as they are uninteresting, and only be required perform operations between nonzero elements. We could thus represent a matrix with n rows and m columns with a partitionable set P of size n with an array P[*Contents*], whose elements are themselves partitionable sets of size m with their own corresponding arrays of integers, so that P[*Contents*][i][*Contents*][j] is the element of the matrix $P(i,j)$. The interesting subsets of the partitionable sets are where we find nonzero elements. Let us call this data structure the *partitionable set matrix*. Since we do not need to worry about nonzero rows of the matrix, we don't need to initialize these rows, so if h is the number of rows with nonzero elements, the data structure takes $O(h \cdot m)$ space. We also now have the ability to skip the nonzero elements of our matrices and have the following algorithm for matrix multiplication.

```
Multiply(A,B,C):  //(A·B = C)
1)C ← Partionable_Set_Matrix(A.Number_Rows(),B.Number_Columns())
2)for(i ← each row of A with nonzero elements)
3)   for(j ← each nonzero element of row i of A)
4)        for(k ← each nonzero element of row j of B)
5)             if(row i of C is all zeroes)
6)                  Add i to subset of P
7)                  Set up row i of A
8)             if(C(i,k) is zero)
9)                  Add i to subset of A[i][partition]
10)                 C(i,k) ← 0
11)            C(i,k) ← A(i,j)·B(j,k) + C(i,k)
12)            if(C(i,k) was changed to 0)
13)                 remove from subset
14)            if(row i was changed to have no zeroes)
15)                 remove i from P and destroy row i
16)       end for
17)  end for
18)end for
```

The algorithm skips to nonzero elements $A(i,j)$ and then finds nonzero elements $B(j,k)$ so that we can add the product $A(i,j)\cdot B(j,k)$ to $C(i,k)$. It can easily be checked from the description of the matrix data structure that each line of the above algorithm can be executed in constant time. If a is the number of nonzero entries in A and b is the number of pairs $(A(i,j), B(j,k))$ where $A(i,j)$ and

$B(i,j)$ are both nonzero, then the innermost loop occurs $O(a + b)$ times, making the overall algorithm take $O(a + b)$ time. It is noteworthy that had we instead used a data structure and algorithm where we initialize upfront all entries of C, there are cases where the above algorithm is asymptotically faster and more space efficient than simply the initialization of matrix C.

While in general the partitionable set trades for time efficiency at the expense of space efficiency, the partitionable set matrix actually uses the properties of the partitionable set to help save space. In fact, we can expand on this idea and use the partitionable set matrix to represent another partitionable set to help alleviate memory costs. If our domain is $\{1, 2, \ldots, n\}$ where $n = m_1 \cdot m_2$, we can represent the partitionable set as a partitionable set matrix with m_1 rows and m_2 columns. An integer $0 \leq i \leq n$ where $i = a \cdot m_1 + r$ for $0 \leq r < m_1$ would then correspond uniquely with entry (a, r) in the matrix. We retain all the constant time operations as before, but now, if q is the number of rows with nonzero entries, the overall memory requirement for the data structure is only $O(m_1 + q \cdot m_2)$, which can save space especially if the interesting elements are heavily distributed in one area and sparse in another. We can also easily extend this procedure to higher dimensional matrices by looking at the value when dividing n by, say, 2^k for several different k, but this would come at a greater time cost for individual operations.

Similarly, we can save memory by implementing a hash table with a partitionable set whose interesting subset are the nonempty locations of the table. This procedure would again have memory benefits over the standard partitionable set but at the cost of non-constant time to locate

specific elements. However, it would have a distinct advantage over other hash tables in that larger sizes of hash table do not negatively impact the time efficiency, only the memory efficiency.

Section II
Graph Representation

Section 2.1: The Adjacency Matrix and Adjacency List

We now turn our attention to the subject of graph representation. In this section and the next, we will be concerned simple, undirected, unweighted graphs. At times we will mention other types of graphs, and the ideas presented here can clearly extend to other types of graph, specifically directed and weighted graphs. We will in generality discuss a graph G with a set of n nodes labeled 1 through n and a set of m edges, which are unordered pairs of nodes. For simplicity, we will allow an edge to connect a node with itself.

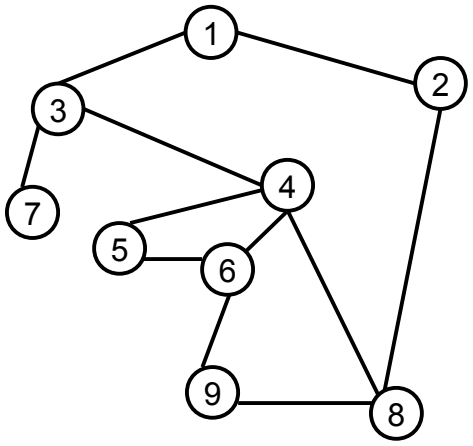

Figure 7: A graph of 9 nodes

There are two data structures typically presented in a discussion on how to represent G in a computer. The first is the adjacency matrix. The adjacency matrix is an $n \times n$ matrix, where matrix element $a_{ij} = 1$ if there is an edge between nodes i and j, and $a_{ij} = 0$ if there is not. In a weighted graph, a_{ij} gets the value of the weight of the edge.

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0	1	0
3	1	0	0	1	0	0	1	0	0
4	0	0	1	0	1	1	0	1	0
5	0	0	0	1	0	1	0	0	0
6	0	0	0	1	1	0	0	0	1
7	0	0	1	0	0	0	0	0	0
8	0	1	0	1	0	0	0	0	1
9	0	0	0	0	0	1	0	1	0

Figure 8: Adjacency matrix for Figure 7

While the adjacency matrix reserves space for each possible pair of nodes and thus each possible edge, the adjacency list reserves space only for the edges actually in G. In the adjacency list, we keep an array A of length n, where index i of A refers to node i, so that at $A[i]$, we keep a linked list of nodes adjacent to node i. The individual elements in the linked list store the information about the edges.

1	2 > 3
2	1 > 8
3	4 > 1 > 7
4	3 > 6 > 5 > 8
5	6 > 4
6	4 > 5 > 9
7	3
8	4 > 9 > 2
9	8 > 6

Figure 9: Adjacency list for Figure 7

The relative efficiency of the two data structures described above will depend on the algorithm to be used. Using the adjacency matrix, we have constant-time access to information about any pair of nodes in G. Thus, we can easily find out if an arbitrary pair (i,j) is an edge and can add or remove edge (i,j) in constant time. However, our algorithm may need to find incident edges to a specified node i, in which case we may need to search through all of row i in order to find it, taking $O(n)$ time. The adjacency list, on the other hand, explicitly keeps track of the incident edges, and so we can locate an arbitrary edge incident to any node i in constant time. This, however, comes at the cost of not being able find a specified pair (i,j) in constant time.

The structure of G further affects the relative efficiency of the two data structures. If the number of edges m is quite small compared to n^2 (the upper asymptotic bound on m), then to find incident edges, the adjacency matrix would likely need to search through many nodes before it

finds and adjacent one. On the other hand, if m is asymptotically close to n^2, in order to locate a specific edge, the adjacency list would likely have to search through many incident edges first.

Section 2.2: Representation of Graphs

Our choice of which of the above data structures depends on the specific graphs and specific algorithms to be used. However, the number of edges and connectivity of the graph might not be known before the choice needs to be made, and we may want to perform operations and algorithms more efficient on one of the data structures in succession with operations and algorithms more efficient on the other.

Our goal, then, is to try to avoid this problem altogether by using a data structure that can locate both specific edges and also arbitrary edges incident on a given node in constant time. As the adjacency matrix is an array of arrays that keep track of all possible edges and the adjacency list is an array of linked lists that keep track of edges actually present, we will use a partitionable set of partitionable sets, precisely the partitionable set matrix described in Section 1.5, to keep track of both possible and actual edges. The interesting elements here are the edges in G. The partitionable set matrix allows for constant-time access to specific pairs and also allows us to locate an interesting pair, one where there is an edge, in constant time.

To compare the efficiency of the partitionable set representation to that of the adjacency matrix and adjacency list, we provide a list of some fundamental graph operations in Table 4.

	Adjacency List	Adjacency Matrix	Partitionable Set
`Set_Number_Of_Nodes(n)` (creates the data structure to accommodate the *n* nodes)	$O(n)$	$O(n^2)$	$O(1)$
`Add_Edge(n1, n2)` (places an edge between nodes *n1* and *n2*)	$O(1)$	$O(1)$	$O(1)$
`Is_Edge(n1, n2)` (returns true if there is an edge between *n1* and *n2*)	$O(n)$ $O(m/n)$ on average	$O(1)$	$O(1)$
`Remove_Edge(n1, n2)` (removes the edge between *n1* and *n2*)	$O(n)$ $O(m/n)$ on average	$O(1)$	$O(1)$
`Find_Any_Incident_Edge(n1, n2)` (given n1, produces a value for n2 such that (n1,n2) is an edge in G)	$O(1)$	$O(n)$	$O(1)$
`Find_Any_Edge(n1, n2)` (Finds some edge in the graph, sets *n1* and *n2* equal to the values of its nodes)	$O(n)$	$O(n)$	$O(1)$
`Number_Of_Nodes()` (returns the number of nodes in the graph)	$O(1)$	$O(1)$	$O(1)$
`Number_Of_Incident_Edges(n)` (returns the number of nodes that share an edge with node *n1*)	$O(1)$	$O(1)$	$O(1)$
`Number_Of_Edges()` (returns the number of edges in the graph)	$O(1)$	$O(1)$	$O(1)$
Overall Space Requirements	$O(n + m)$	$O(n^2)$	$O(n^2)$

Table 4: Efficiency of various operations for graph G

The operations listed in Table 4 should all be straightforward to implement from the descriptions given thus far. It should be noted that `Remove_Edge(n1, n2)` as listed in the table for the adjacency list assumes a general and perhaps somewhat naïve implementation. If we are working with directed graphs, and always call `Find_Any_Incident_Edge(n1, n2)` before we call `Remove_Edge(n1, n2)`, then `Remove_Edge(n1, n2)` could always in this case be performed in constant time. In undirected graphs, we could strengthen the adjacency list by

storing information at each pair (i,j) a pointer to the location in the list of pair (j,i). Then the same conclusion as above would hold.

The last three operations in the table might not necessarily be a part of the data structures as described, but they can be implemented using only integers as a counters. As such, they are trivial operations and meaningless in the further discussion of efficiency but will be used in the algorithms presented.

As can be seen from the graph, the asymptotic time efficiency of the partitionable set representation dominates that of the adjacency list and adjacency matrix. We will then try to argue that the list of operations in Table 4 is essentially complete. In other words, using these operations, we can perform any algorithm we would want to perform using the adjacency matrix or adjacency list. Furthermore, if the partitionable set representation is used to implement the operations, these algorithms will be performed in the same asymptotic time. We show this by demonstrating that the operations in Table 4 can by themselves be used to represent an adjacency matrix or an adjacency list.

To represent an $n \times n$ matrix using the operations from table 4, we only need to know that we can locate and change the contents in constant time of any entry in the matrix. To create an $n \times n$ matrix, then, we need to create a graph of n nodes. To locate and possibly change the contents of

entry (*i*,*j*) in the matrix, we can call the corresponding operation `Add_Edge(n1, n2)`, `Is_Edge(n1, n2)`, or `Remove_Edge(n1, n2)` with *n1* = *i* and *n2* = *j*. These are constant-time operations under the partitionable set, so we have indeed represented a matrix with constant-time operations.

Unfortunately, an array of linked lists cannot be truly represented by only the operations in Table 4. As a counterexample, we can take the linked lists at indices i and j and concatenate them in constant time, placing the result in one of the two indices, leaving the other empty. Such an operation would have virtually no practical use in an adjacency list data structure. While one might view this operation as combining two nodes into one, it only addresses outgoing edges and not incoming ones. Combining incoming edges would be quite inefficient on an adjacency list data structure. Secondly, it would be hazardous to do this without first knowing that the two lists do not have an element in common, but checking this property first would put this operation on the same asymptotic time as can be done with the operations in Table 4. Nonetheless, this leaves open the possibility that the adjacency list can perform some algorithm in faster asymptotic time than on the partitionable set. That being said, we can successfully implement all the algorithms in [1] without any loss of asymptotic time efficiency.

The functionality of the adjacency list that we will try to represent is that we should be able to add and remove edges, find an arbitrary incident edge, and cycle through edges incident on a

given node, each in constant time. Using the above operations, we could use the `Find_Any_Incident_Edge(n1, n2)` and `Remove_Edge(n1, n2)` operations in the following way to cycle through edges.

```
G.Cycle_Through_Incident_Edges_1(n1):
1)S ← Stack_Of_Integers
2)n2 ← Integer
3)while(G.Number_Of_Incident_Edges() > 0)
4)    G.Find_Any_Incident_Edge(n1, n2)
5)    G.Remove_Edge(n1, n2)
      ...
      //Do something with edge
      ...
6)    S.Push(n2)
7)end while
8)while(S.Size() > 0)
9)    S.Pop(n2)
10)   G.Add_Edge(n1, n2)
11)end while
```

If we have k edges incident on node *n1*, then the previous algorithm cycles through these edges in $O(k)$ time. However, doing things this way presents us with two problems. First, while we should expect to be able to cycle through edges on each of the data structures using a constant amount of memory, namely the amount of memory needed to store one integer. Using the stack to store the already checked adjacent nodes, we use $O(k)$ extra memory. If the algorithm we wish

to implement has to cycle through all the edges in *G*, we practically have to create an entirely new data structure to store all the edges of *G*. Furthermore, we may want to be able to access edges while we are cycling through them. Thus, we probably do not want to destroy the data structure by removing edges if we do not have to.

One way we can fix this problem is by changing the requirements on the add and find operations to allow for a first-in, first-out add and removal of edges representative of a linked list data structure. The partitionable set can accommodate this functionality, as described in Section 1.5 by having the contents array of the partitionable set be an array of pointers into a linked list data structure. We then can add, find, and remove edges as we normally would with a linked list without negating the constant-time access to specific edges. Thus, we can cycle through edges in the following manner.

```
G.Cycle_Through_Incident_Edges_2(n1):
1)for (i ← 1; i < G.Number_Of_Incident_Edges(n1); i++)
2)    n2 ← Integer
3)    G.Find_Any_Incident_Edge(n1, n2)
4)    G.Remove_Edge(n1, n2)
      ...
      //Do something with edge
      ...
5)    G.Add_Edge(n1, n2)
6)end for
```

Layering the partitionable set on a linked list gives the resulting data structure all the functionality of the adjacency list as well as the adjacency matrix. However, adding this linked list structure would add considerable time to the operations (which should be clear later), so doing so is perhaps more theoretically interesting than practically interesting. In practice, we probably do not need to specifically have a linked list structure in order for an algorithm to function properly. Rather, we only need to list incident edges in some arbitrary order.

If we relax our previous goal of trying to fully represent an adjacency list, getting around the problems present in `Cycle_Through_Incident_Edges_1(n1)` is easy. We can introduce an iterator operation to Table 4, and the partitionable set can implement this operation using its own iterator as introduced at the end of Section 1.3.

We can also introduce some operations that are together much stronger than an iterator. Currently, there are only two sets of pairs of nodes with which we are concerned, edges and non-edges. We could further partition the set of edges into examined and unexamined edges in the algorithm we are performing. Where we already have operations for adding and removing edges and finding the number edges, we add the analogous operations for the examined and unexamined subsets of edges. We can easily support this functionality with a partitionable set with two interesting subsets as described in Section 1.3. We can thus cycle through edges in the following way.

```
G.Cycle_Through_Incident_Edges_3(n1):
1)n2 ← Integer
2)while(G.Number_Of_Unexamined_Edges() > 0)
3)   G.Find_Any_Unexamined_Edge(n1, n2)
4)   G.Examine_Edge(n1, n2)
     ...
     //Do something with edge
     ...
5)end while
6)G.Unexamine_All_Edges(n1)
```

This third method is quite similar to method 1, but we no longer have the extra memory cost. The graph operations in lines 2, 3, and 4 of `Cycle_Through_Incident_Edges_3(n1)` are all analogous to lines 3, 4, and 5 of `Cycle_Through_Incident_Edges_1(n1)`. The `Unexamine_All_Edges(n1)` operation in line 6 can be done in constant time by taking the union of the examined and unexamined subsets, which is described in a little more detail at the end of Section 1.3. In fact, we could also have an operation that unexamines all the edges in *G* in constant time, if we keep track of the size of the examined edge subsets using another partitionable set.

We now show that these newly introduced operations are a bit stronger and offer more control over the edges in *G* than the previous methods. Consider the following algorithm, which

produces an eulerian cycle, a path that goes through each edge exactly once and ends exactly where it started, in an undirected graph, assuming such a path exits.

```
G.Produce_Eulerian_Cycle():
1)P ← path
2)while(G.Number_Of_Examined_Edges() > 0)
3)    n1, n2 ← Integer
4)    G.Find_Any_Unexamined_Edge(n1,n2)
5)    Add (n1, n2) to P
6)    Examine_Edge(n1, n2)
7)    while(n2 ≠ n1)
8)          n3 ← Integer
9)          G.Find_Any_Unexamined_Incident_Edge(n2, n3)
10)         G.Examine_Edge(n2, n3)
11)         Add (n2, n3) to P
12)         n2 ← n3
13)   end while
14)end while
15)G.Unexamine_All_Edges()
```

We will not go into detail about how to represent the eulerian path *P* or how to add edges to it, as that is unimportant to the current discussion. We will also not prove the correctness of this algorithm. What is important, however, is that the algorithm constructs arbitrary paths in *G* until it finds a cycle, and it joins these cycles together as it constructs *P* . The critical issue here is that once we locate an edge, we want to ignore it and skip it for the rest of the algorithm. We could do

this by removing the edge completely from the graph, but in general doing so would leave us with the same problems as discussed in `Cycle_Through_Incident_Edges_1(n1)`. Having an iterator or changing the remove and find operations to have first-in, first-out functionality would also not solve this problem.

The two previous methods of cycling through incident edges do not have to be mutually exclusive methods, as the method of using examined and unexamined edges would take longer than using an iterator but also provides more functionality. The partitionable set could offer some other straightforward additions to the operations of Table 4 as well, such as an operation that locates in constant time a pair of nodes that do not share an edge by using the variant of partitionable set described in Section 1.4, but to simply implement an algorithm in the same way an adjacency matrix or adjacency list would, these additions are unnecessary.

Section III
Comparison of the Data Structures

Section 3.1: Methodology of the Testing

As shown in Table 4, the timings of operations implemented using the partitionable set data structure asymptotically dominate those of operations implemented using either of the other two data structures. This is of course not the entire story, and one might wonder what the constants associated with the asymptotic bounds are. In this section we will present some timings of algorithms that should help demonstrate what these constants are and what their effect is while running practical algorithms.

While we reference the operations in Table 4 to facilitate the discussion of the relative efficiency of the three data structures, we did not implement the algorithms in the next section using this abstraction. Instead, we implemented the algorithms on the different data structures from the ground up to ensure that the timings truly reflect the fastest possible implementation the data structure can provide. For the partitionable set data structure, we elected not to have a partitionable set of partitionable sets but rather an array of partitionable sets, since all the graphs we tested have for every node an edge incident on it, so we don't get any benefit from not initializing the node or being able to skip uninteresting ones. It would only have increased the time of all the other operations. Also, in practice we may not ever need to use the operation or to

have run in constant time rather than $O(n)$ time, and we did not specifically test the performance of these operations here.

We tested the following algorithms on graphs with various numbers of edges. One example of a graph we tested was one with a node for each space on an $n \times m$ chessboard, and two nodes share an edge if a knight chess piece can legally travel from that space on the board to the other, so each node is adjacent to at most eight others. We chose to test this graph because it can be easily and deterministically constructed and its edges are distributed somewhat predictably and uniformly. We also created graphs formed by taking the union of smaller complete graphs and connecting the complete sub-graphs with one edge. Thus, if our graph has n nodes, each node would share an edge with approximately $n/8$ other nodes. Graphs with randomly added edges were tested as well. While the aforementioned graphs have no real-world applicability and the output of the algorithm can for the deterministic graphs be easily deduced beforehand, the timings of the algorithms on these graphs should be fairly indicative of timings of algorithms performed on more practical graphs that have a similar number of edges. At the very least, they should provide a good indication of what the constants associated with the operations in Table 4 are.

We tested both graphs that have a number of edges on the order of the number of nodes (e.g. the knight graph) and graphs that have a number of edges on the order of the square of the number of nodes (e.g. the union of complete graphs). The adjacency list would be expected to perform at its

relative best on the most sparse of graphs. In other words the graph of n nodes would have $O(n)$ edges. On the other hand, the adjacency matrix would be expected to perform at its relative best on the most dense of graphs, those graphs with n nodes and $O(n^2)$ edges. Thus, if the partitionable set performs better than the adjacency list and adjacency matrix on both graphs with $O(n)$ edges and those with $O(n^2)$ edges, then we can conclude that it should perform better than the other two on graphs with a number of edges between those two bounds and thus virtually any graph it can represent.

Section 3.2: Timing Comparison

In this section we will look at the timings of various algorithms on the three data structures presented.

Section 3.2.1: Graph Construction

We start by analyzing the construction of the graph itself, namely the process of putting the representation of the graph into memory. By looking at the timings presented in this section, we should get a better idea of the efficiency of the `Set_Number_Of_Nodes(n)`, `Add_Edge(n1, n2)`, `Is_Edge(n1, n2)`, and `Remove_Edge(n1, n2)` operations from Table 4.

Figures 10 and 11 list the timings of constructing the graph in the most straightforward way. We know a priori the number of nodes that will be in the graph once we start constructing it, and then we simply read off the list of edges to be added to the graph. Thus, we only need to call `Set_Number_Of_Nodes(n)` and then `Add_Edge(n1, n2)` for each edge.

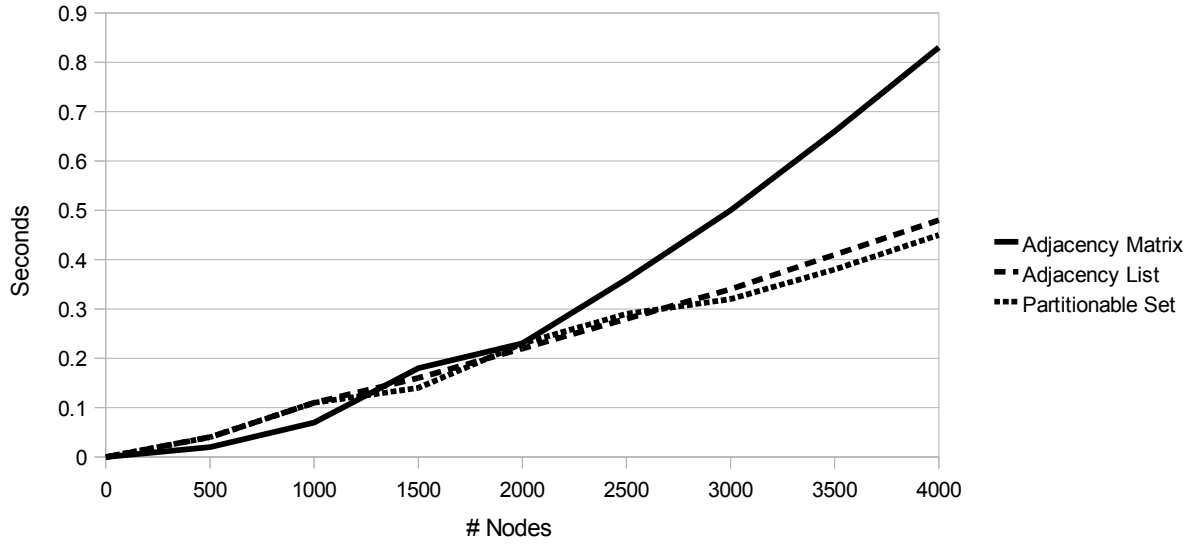

Figure 10: Constructing a graph with ~8 edges per node

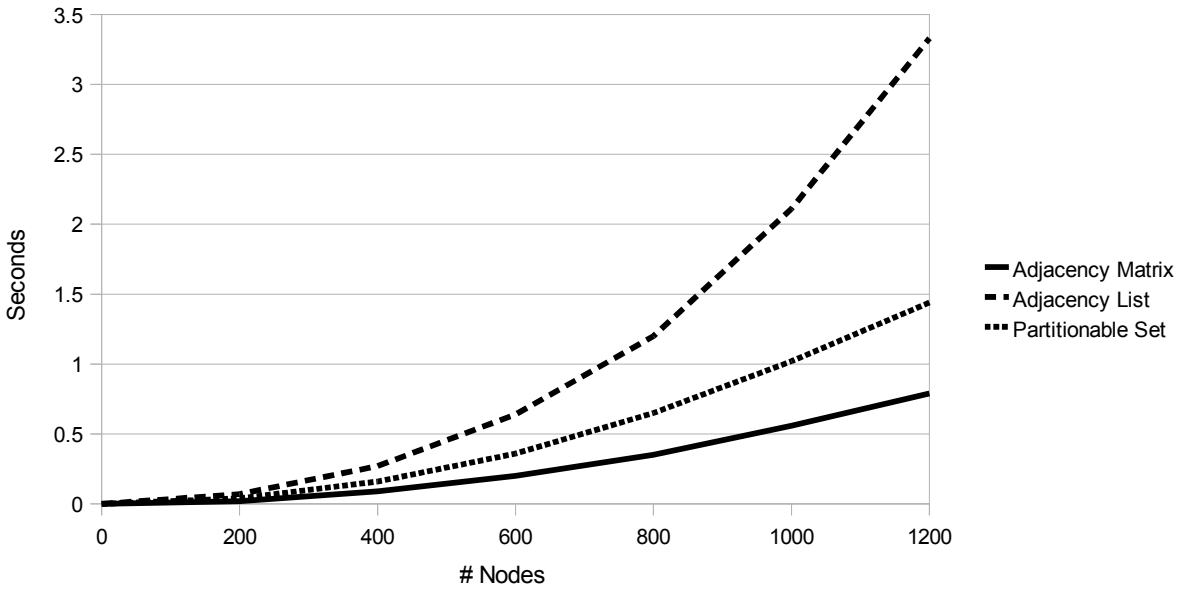

Figure 11: Constructing a graph with ~n/8 edges per node

For graphs with $O(n^2)$ edges, the adjacency matrix performs the best of the three, showing that it has the most efficient for adding edges. However, for sparser graphs, as seen in Figure 10 with $O(n)$ edges, the $O(n^2)$ running time of creating the matrix to support n nodes gets in the way.

On the other hand, the adjacency list performs very poorly for dense graphs with $O(n^2)$ edges due to a high cost of adding edges but does well for sparse graphs. For the most sparse of graphs, the adjacency list and partitionable set representations perform about equivalent to one another. However, there is one more expense we might want to consider in analyzing the time needed to create a graph, and that is the cost of destroying the graph, where we free up the memory used by the representation. Since the adjacency matrix and partitionable set are made up of large

contiguous blocks of memory, freeing this up is relatively easy. However, the adjacency list must look at all the pointers used in the linked list in order to destroy them, taking $O(m)$ time. Figure 12 shows the time needed to do this for the same graphs as in Figure 10. For reference, the partitionable set took .04 seconds to destroy a graph of 4000 nodes.

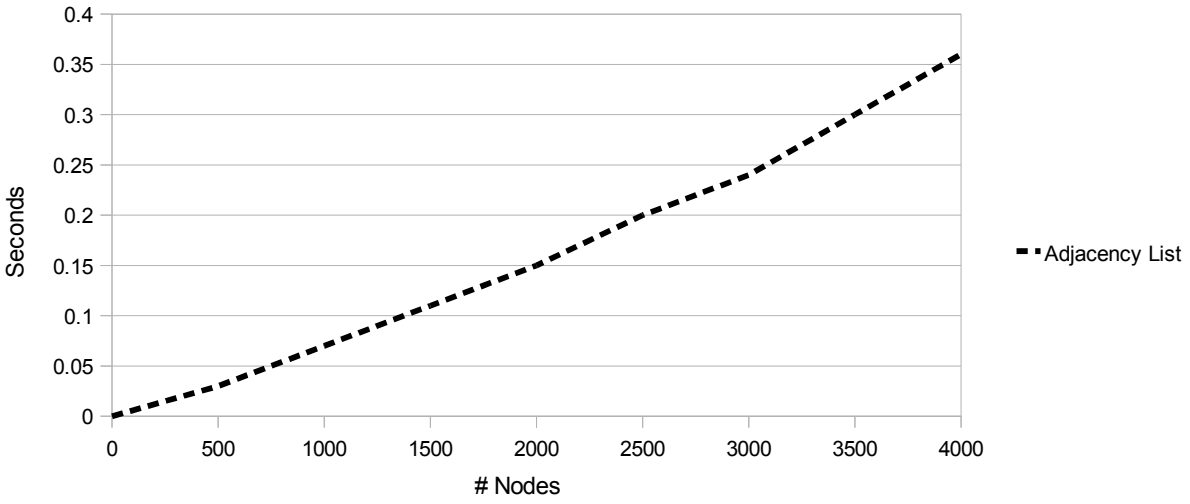

Figure 12: Destroying a graph with ~ 8 edges per node

Hence, we can expect the partitionable set to be able to construct graphs as fast as the adjacency list for virtually any type of graph. If we choose not to ignore the time to destroy the graph when we are done with it, the partitionable set is strictly faster. For dense and small graphs, we might still expect the adjacency matrix to perform better than the partitionable set, but if the number of edges is small enough compared to n^2, the partitionable set should do better.

When constructing graphs, we might also need to consider the issue of the possibility of accidentally adding the same edge more than once. If we are adding randomly generated edges to the graph or if the input from which we are constructing the graph is likely to contain repeats of edges, we probably first want to check if an edge already exists in the graph before adding it. Otherwise, we could get faulty behavior in an algorithm later run on this graph. Figure 13 shows the timings of constructing the same graphs as in Figure 10, but now we call `Is_Edge(n1, n2)` before adding the edge.

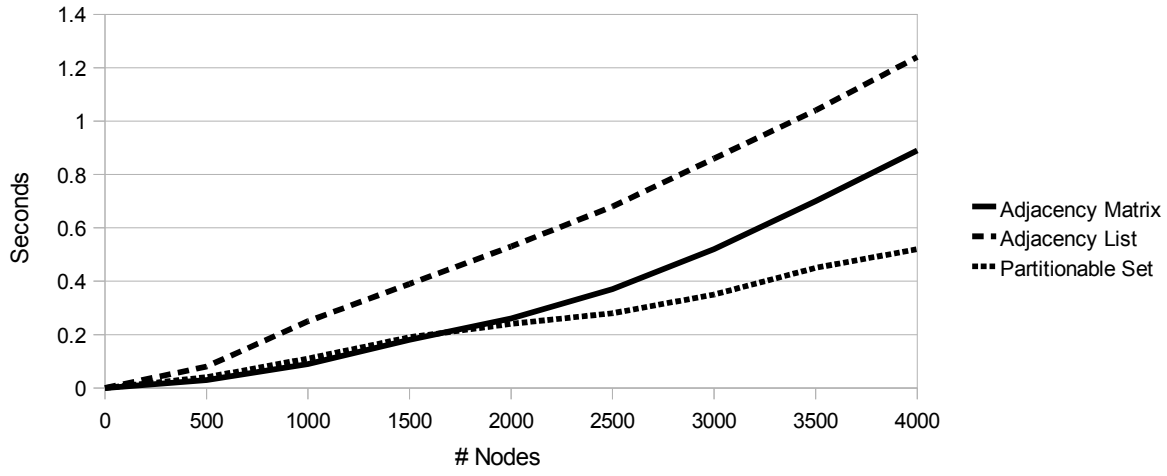

Figure 13: Creating a graph of ~8 edges per node with checking

Comparing Figure 13 to Figure 10, the difference `Is_Edge(n1, n2)` adds to the adjacency matrix and partitionable set is quite small. However, even for sparse graphs, the operation adds

quite a bit of time to the adjacency list implementation. For denser graphs, the difference is too large to be practically implemented on the adjacency list.

Finally, Figure 14 shows the process of removing all the edges of a graph. While not a practical algorithm, it shows the relative efficiency of `Remove_Edge(n1, n2)` so that we can see what kind of effect repeated calls to that operation have in the long term.

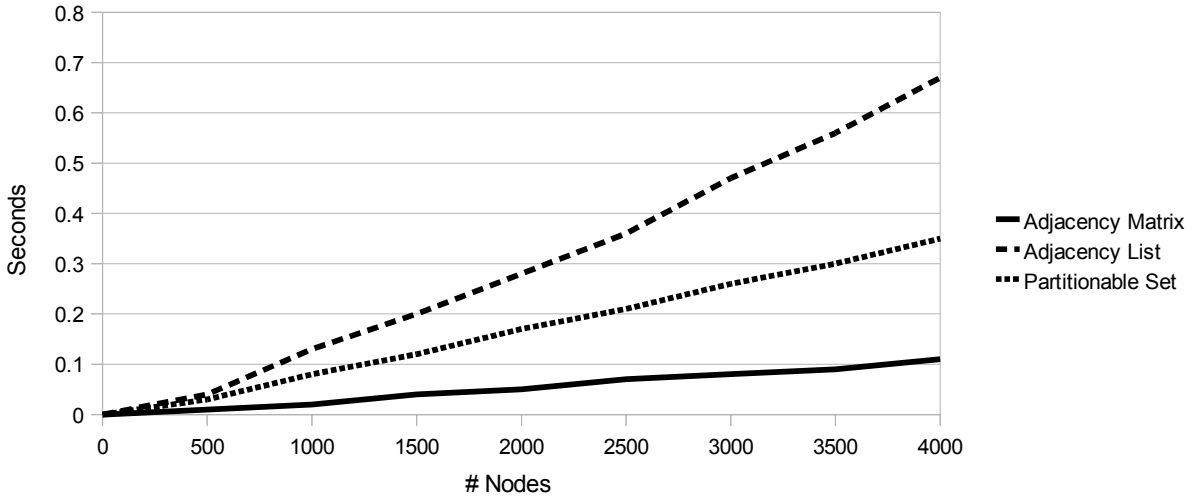

Figure 14: Removing edges from a graph with ~ 8 edges per node

Again, the adjacency matrix is the most efficient, and for denser graphs the operation is impractical for extensive use on the adjacency list.

Section 3.2.2: Breadth-first Search

Now that we have looked at creating the graph, we now consider an actual algorithm we might want to implement: breadth-first search. We give a version of breadth-first search below that uses the extension of the graph operations from Section 2.2 where we can examine and unexamine edges.

```
Breadth_First_Search(G, start_node):
1)Q ← queue whose only element is start_node
2)while(Q is not empty)
3)    n1 ← Dequeue(Q)
4)    for(n2 ← each unexamined node adjacent to n1)
5)         if(not examined[n2])
6)              Examine_Edge(n1, n2)
7)              Enqueue(Q,n2)
8)         end if
9)    end for
10)end while
11)Unexamine_All_Nodes(G)
```

The algorithm as shown does not produce anything, but it does show the behavior of breadth-first search that is important to this discussion. The critical line here is line 5. While lines 6 and 11 may be less efficient depending on the representation, we could implement breadth-first search using an array that tells us which nodes have been examined rather than which edges, which would mean line 5 would be the only line that interacts directly with the representation. Figures

15 and 16 show the timings of the algorithm with the implementation using the examined node array. On line 5, we iterate through all the edges incident on a given node, and so the differences in the timings seen in figures 15 and 16 are entirely based on how quickly the representation can iterate through incident edges.

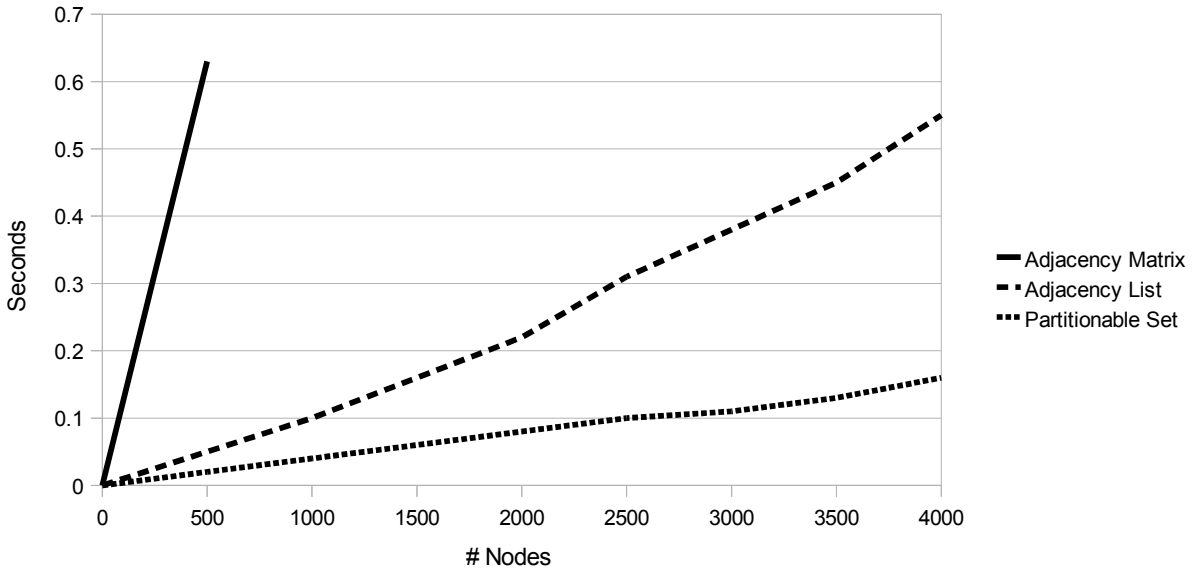

Figure 15: Breadth-first search on a graph with ~8 edges per node

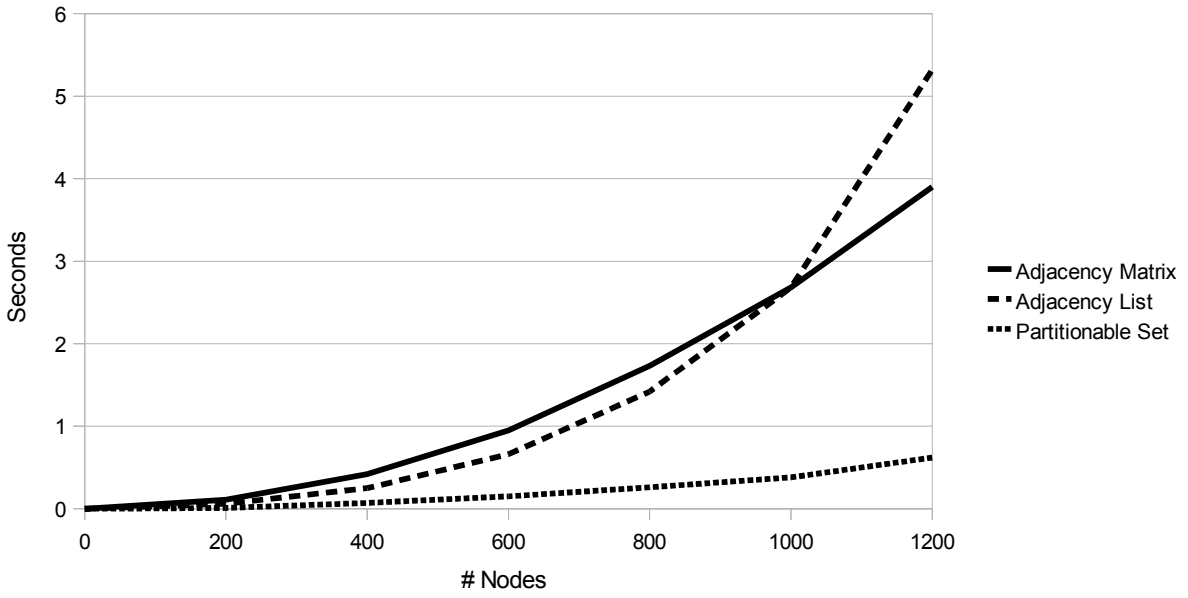

Figure 16: Breadth-first search on a graph with ~ n/8 edges per node

For both sparse and dense graphs, the partitionable set performs quite a bit better than either of the other two representations. This result should be fairly intuitive, as iterating through incident edges requires only iterating through an interesting subset as described in Section 1.3, which at each iteration only requires incrementing a counter and returning the value stored in the array *Partition*.

There are of course many algorithms we could test besides just breadth-first search. However, most graph traversal algorithms will have an iterator through a node's incident edges as its primary interaction with the graph representation. Hence, many of the same conclusions we can

draw from looking at breadth-first search can be drawn from other graph traversal algorithms like depth-first search and minimum spanning tree algorithms.

Section 3.2.3: Floyd-Warshall Algorithm

In section 3.2.2 we considered breadth-first search and more generally graph traversal algorithms, which are intended primarily for use on an adjacency list representation, as can perhaps be inferred from Figures 15 and 16. However, there are also some algorithms that make explicit use of the adjacency matrix representation, and we give an example of the Floyd-Warshall algorithm, as described in [1], for finding the transitive closure of a graph (or, more generally, the distance of any node to any other). We give the pseudocode below, which operates directly on a matrix data structure whose values (i,j) are true or false depending on whether there is a path from node i to node j.

```
Transitive_Closure_1(G):
1)n ← number of nodes in G
2)A ← n×n boolean matrix
3)for (each pair (i,j) in A)
4)   A(i,j) ← true iff (i,j) is an edge in G
5)end for
6)for (k ← 1 to n)
7)   for (i ← 1 to n)
8)        for (j ← 1 to n)
9)             A(i,j) ← A(i,j) or (A(i,k) and A(k,j))
10)       end for
11)  end for
12)end for
13)return A
```

At each iteration of line 9, we check if we have already found a path from node *i* to node *j* or if there is a path from *i* to *j* through node *k*. Since we are checking specific pairs of nodes, this is clearly not an algorithm we would want to implement using an adjacency list rather than the matrix *A*. However, as stated earlier, the partitionable set representation can perform every matrix operation in constant time. We could perhaps replace *A* in this algorithm with another instance of a graph represented by a partitionable set, but a partitionable set representing a matrix would not be as efficient as having an array of arrays. However, we could use edit the algorithm to benefit from the partitionable set's ability to quickly find incident edges.

```
Transitive_Closure_2(G):
1)n ← number of nodes in G
2)H.Set_Number_Of_Nodes(n)
3)for (each edge (i,j) in G)
4)   H.Add_Edge(i,j)
5)end for
6)for (k ← 1 to n)
7)   for (i ← 1 to n)
8)        for (j ← each integer such that (k,j) is in H)
9)             if (H.Is_Edge(i,k) and not H.Is_Edge(i,j))
10)                 H.Add_Edge(i,j)
11)            end if
12)       end for
13)  end for
14)end for
15)return A
```

Using the partitionable set, we can change lines 3 and 4 of `Transitive_Closure_1(G)` to only examine and copy edges that are in *G*. Line 9 only affects anything if (*i*,*j*) is not in *A* and if (*i*,*k*) and (*k*,*j*) both are. Thus, we can change lines 8 and 9 to skip past instances where (*k*,*j*) is not an edge. `Transitive_Closure_2(G)` shows the result. We could also easily expand on this idea and use a partitionable set where both subsets are interesting to be able to skip past instances where (*i*,*j*) is already an edge when *H* starts getting dense. This previous addition was not tested, however.

Figure 17 shows the result of timing `Transitive_Closure_1(G)` against `Transitive_Closure_2(G)` for graphs of 100 nodes with varying distribution of edges.

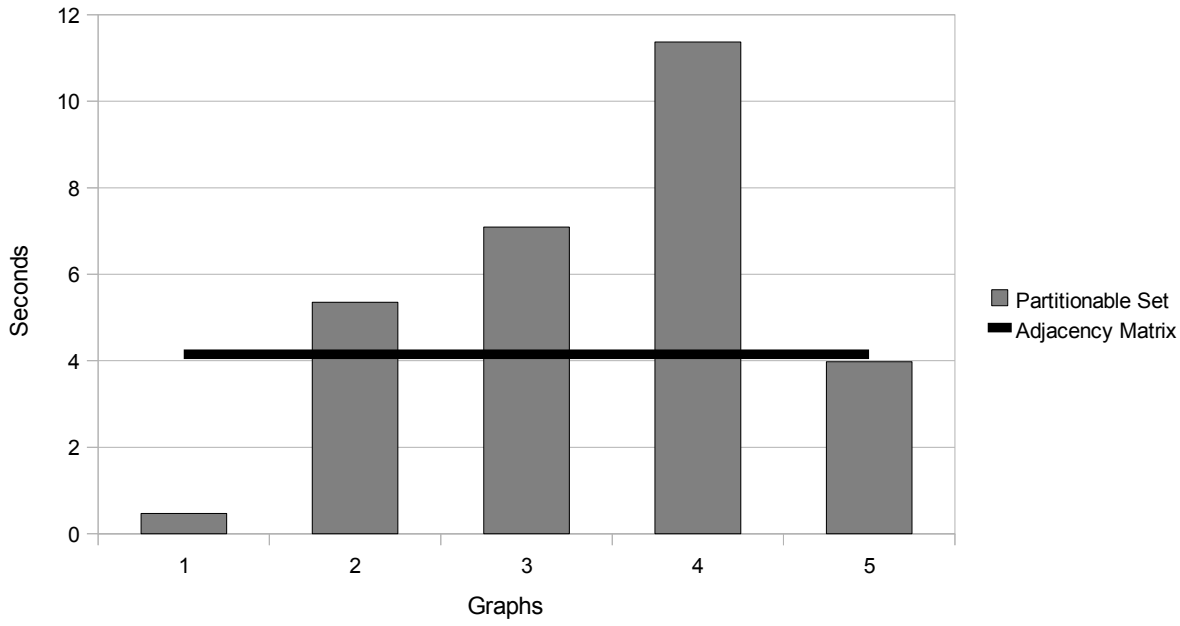

Figure 17: Floyd-Warshall on 100 node graphs

Graphs 1, 2, 3, and 4 were constructed by taking the union of 8, 4, 2, and 1 complete graphs, respectively. Thus, the final result of the algorithm on graphs 1, 2, 3, and 4 has A or H with $n/8$, $n/4$, $n/2$, and n interesting elements per node. Graph 4, the complete graph, shows the worst case relative running time of the partitionable set against the adjacency matrix, since we do not get any benefit from skipping past non-edges, as there are none. As seen in Figure 5 this graph shows us an upper bound of about 3 on the relative inefficiency of the partitionable set against

the adjacency matrix. On the other hand, there is no such constant bound in the other direction, and for graphs that are less connected that Graph 1, we can expect partitionable set to be much more efficient. Graph 5 is a line, where there are only edges of the form (i, i + 1). The end result of the algorithm for Graph 5 is the same is in Graph 4, but there are fewer edges in the beginning, thus showing an example of the effect the initial number of edges has on the overall efficiency of the algorithm.

Section IV
Conclusions and Possibilities for Further Work

Section 4.1: Time Efficiency

If we are concerned only with time efficiency, the partitionable set wins out over the other two representations overall. In virtually every test graph and algorithm, the partitionable set performed faster than the adjacency list and as such would be a good choice for whatever algorithm we may have desired beforehand to implement using an adjacency list.

Against the adjacency matrix, the partitionable set did not dominate across all test cases. For small graphs and graphs of n nodes with $O(n^2)$ edges, the adjacency matrix may perform better than the partitionable set for simple operations such as locating edges or for matrix-specific graph algorithms. For every other type of graph, even under the aforementioned operations and algorithms, the partitionable set appears to be the better choice.

Overall, the partitionable set is perhaps the best choice for a general-purpose component, if multiple algorithms intended for two different data structures will be used, or if the number of edges in the graph is not known beforehand.

Section 4.2: Memory Efficiency

Unfortunately, the partitionable set is not so efficient when it comes to memory requirements. It uses the more memory than either of the other two representations. This difference may not be so important when comparing the partitionable set against the adjacency matrix, but while the partitionable set is much faster than the adjacency list, the memory requirements needed to store a sparse graph may so much greater on the partitionable that they negate its time efficiency benefit. One reason is that many problems may be solvable using the $O(n + m)$ memory requirement from the adjacency list but completely impractical using the $O(n^2)$ memory of the partitionable set. Another reason is that in practice the extra storage requirement actually does lead to time inefficiencies due to longer access and allocation time. Further testing may help establish what we can expect these extra inefficiencies to be.

To fix this memory issue, we could try to use the techniques described at the end of Section 1.5. Instead of representing a graph as a partionable set of partitionable sets, we could represent it as a partitionable set of 2-dimensional $n^{1/2} \times n^{1/2}$ partitionable set matrices. Then if, for example, we want to represent a class of graphs where each node is connected to at most some $O(n^{1/2-\varepsilon})$ other nodes for some $0 < \varepsilon \leq 1/2$, then each element of the inner partitionable set matrix will use $O(n^{1/2} + n^{1/2-\varepsilon} \cdot n^{1/2}) = O(n^{1-\varepsilon})$ space. The entire graph would then require only $O(n^{2-\varepsilon})$ space rather than the $O(n^2)$ space as before.

If we further know that the edges are heavily distributed in a certain area or have a predictable distribution so that we can manage to only use c different rows of the inner partitionable set matrix, we can represent the graph using $O(n^{3/2})$ space even if there are as many as $O(n^{1/2})$ edges per node. It is reasonable to predict that in practice we might have nodes adjacent to similarly numbered nodes in the representation. If we know that each node i is adjacent mostly to nodes numbered in a tight interval around i, we may even be able to use a k-dimensional partitionable set instead of a 2-dimensional partitionable set to get the space requirement down to as little as $O(n^{1+1/k})$. Using multidimensional partitionable set matrices would decrease the time efficiency, but judging from the timings of algorithms as in Section 3, we might be able to use 2, 3, or perhaps 4-dimensional matrices and still have algorithms run as fast as they would under the adjacency list. Further testing is needed on real-world applications to determine what the exact costs and benefits of this method would be and whether it would even be practically implemented in order to save memory.

A simpler method would be to use a partitionable set in the implementation of a hash function on the adjacent nodes. A hash function by itself would already be a decent choice to alleviate some of the problems in choosing whether to use an adjacency matrix or adjacency list representation, as it can be viewed as a sort of hybrid between the two, layering linked lists inside of arrays to distributing the costs of access to specific edges with the costs of iterating through incident edges. If we view a regular hash as a hybrid of an adjacency list and adjacency matrix, then a hash function built on top of a partitionable set would be a hybrid of the adjacency list and

partitionable set representations, distributing the memory costs with the costs of access to specific edges. Again, further analysis of real-world problems would clarify the costs and benefits of this method.

References

[1] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2nd ed. Cambridge, MA: The MIT Press, 2003. Print.